The implementation

Parse time and compile time

Virtual machine

The parser

The Factor parser reads textual representations of objects and definitions, with all syntax determined by **Parsing words**. The parser is implemented in the **parser** vocabulary, with standard syntax in the **syntax** vocabulary. See **Syntax** for a description of standard syntax.

The parser cross-references **Source files** and **Definitions**. This functionality is used for improved error checking, as well as tools such as **Definition cross referencing** and **Editor integration**.

The parser can be invoked reflectively, to run strings and source files.

Evaluating strings at run time

run-file (file --)

parse-file (file -- quot)

If Factor is run from the command line with a script file supplied as an argument, the script is run using **run-file**. See **Command line arguments**.

While **run-file** can be used interactively in the listener to load user code into the session, this should only be done for quick one-off scripts, and real programs should instead rely on the automatic **Vocabulary loader**.

See also

Parsing words, **Definitions**, **Definition sanity checking**

Evaluating strings at run time

The **eval** vocabulary implements support for evaluating strings of code dynamically.

The main entry point is a parsing word, which wraps a library word:

eval(

eval (str effect --)

This pairing is analogous to that of **call(** with **call-effect**.

Advanced features:

Evaluating strings with a different vocabulary search path

eval>string (str -- output)

Definitions

A *definition* is an artifact read from a source file. Words for working with definitions are found in the **definitions** vocabulary.

Definitions are defined using parsing words. Examples of definitions together with their defining parsing words are words (**:**), methods (**M:**), and vocabularies (**IN:**).

All definitions share some common traits:

There is a word to list all definitions of a given type

There is a parsing word for creating new definitions

There is an ordinary word which is the runtime equivalent of the parsing word, for introspection

Instances of the definition may be introspected and modified with the definition protocol

For every source file loaded into the system, a list of definitions is maintained. Pathname objects implement the definition protocol, acting over the definitions their source files contain. See **Source files** for details.

Definition protocol
Definition sanity checking
Compilation units

A parsing word to remove definitions:

FORGET:

See also

Printing definitions, **The parser**, **Source files**, **Words**, **Generic words and methods**, **Help system implementation**

Definition protocol

A common protocol is used to build generic tools for working with all definitions.

Definitions must know what source file they were loaded from, and provide a way to set this:

where (defspec -- loc)

set-where (loc defspec --)

Definitions can be removed:

forget (defspec --)

Definitions must implement a few operations used for printing them in source form:

definer (defspec -- start end)

definition (defspec -- seq)

See also

Printing definitions

Definition sanity checking

When a source file is reloaded, the parser compares the previous list of definitions with the current list; any definitions which are no longer present in the file are removed by a call to **forget**.

The parser also catches forward references when reloading source files. This is best illustrated with an example. Suppose we load a source file `a.factor`:

```
USING: io sequences ;
IN: a
: hello ( -- str ) "Hello" ;
: world ( -- str ) "world" ;
: hello-world ( -- ) hello " " world 3append print ;
```

The definitions for `hello`, `world`, and `hello-world` are in the dictionary.

Now, after some heavily editing and refactoring, the file looks like this:

```
USING: make ;
IN: a
: hello ( -- ) "Hello" % ;
: hello-world ( -- str ) [ hello " " % world ] "" make ;
: world ( -- ) "world" % ;
```

Note that the developer has made a mistake, placing the definition of `world` *after* its usage in `hello-world`.

If the parser did not have special checks for this case, then the modified source file would still load, because when the definition of `hello-world` on line 4 is being parsed, the `world` word is already present in the dictionary from an earlier run. The developer would then not discover this mistake until attempting to load the source file into a fresh image.

Since this is undesirable, the parser explicitly raises a **no-word** error if a source file refers to a word which is in the dictionary, but defined after it is used.

The parser also catches duplicate definitions. If an artifact is defined twice in the same source file, the earlier definition will never be accessible, and this is almost always a mistake, perhaps due to a bad choice of word names, or a copy and paste error. The parser raises an error in this case.

redefine-error (definition --)

Compilation units

A *compilation unit* scopes a group of related definitions. They are compiled and entered into the system in one atomic operation.

When a source file is being parsed, all definitions are part of a single compilation unit, unless the **<<** parsing word is used to create nested compilation units.

Words defined in a compilation unit may not be called until the compilation unit is finished. The parser detects this case for parsing words and throws a **staging-violation**. Similarly, an attempt to use a macro from a word defined in the same compilation unit will throw a **transform-expansion-error**. Calling any other word from within its own compilation unit throws an **undefined** error.

This means that parsing words and macros generally cannot be used in the same source file as they are defined. There are two means of getting around this:

The simplest way is to split off the parsing words and macros into sub-vocabularies; perhaps suffixed by `.syntax` and `.macros`.

Alternatively, nested compilation units can be created using **Parse time** .

evaluation

Parsing words which create new definitions at parse time will implicitly add them to the compilation unit of the current source file.

Code which creates new definitions at run time will need to explicitly create a compilation unit with a combinator. There is an additional combinator used by the parser to implement **Parse time evaluation**.

with-compilation-unit (quot --)

with-nested-compilation-unit (quot --)

Additional topics:

Compilation units internals

Vocabularies

A *vocabulary* is a named collection of **Words**. Vocabularies are defined in the **vocabs** vocabulary.

Vocabularies are stored in a global hashtable:

dictionary

Vocabularies form a class.

vocab (vocab-spec -- vocab)

vocab? (object -- ?)

Various vocabulary words are overloaded to accept a *vocabulary specifier*, which is a string naming the vocabulary, the **vocab** instance itself, or a **vocab-link**:

vocab-link

>vocab-link (name -- vocab)

Looking up vocabularies by name:

vocab (vocab-spec -- vocab)

Accessors for various vocabulary attributes:

vocab-name (vocab-spec -- name)

vocab-main (vocab-spec -- main)

vocab-help (vocab-spec -- help)

Looking up existing vocabularies and creating new vocabularies:

vocab (vocab-spec -- vocab)

child-vocabs (vocab -- seq)

create-vocab (name -- vocab)

Getting words from a vocabulary:

vocab-words (vocab-spec -- words)

words (vocab -- seq)

all-words (-- seq)

words-named (str -- seq)

Removing a vocabulary:

forget-vocab (vocab --)

See also

Words, **Vocabulary loader**, **Parse-time word lookup**

Source files

Words in the **source-files** vocabulary are used to keep track of loaded source files. This is used to implement **Runtime code reloading**.

The source file database:
source-files

The class of source files:
source-file (path -- source-file)

Words intended for the parser:
record-checksum (lines source-file --)

record-definitions (file --)

Removing a source file from the database:
forget-source (path --)

Updating the database:
reset-checksums (--)

The **pathname** class implements the definition protocol by working with the corresponding source file; see **Definitions**.

Optimizing compiler

Factor includes two compilers which work behind the scenes. Words are always compiled, and the compilers do not have to be invoked explicitly. For the most part, compilation is fully transparent. However, there are a few things worth knowing about the compilation process.

The two compilers differ in the level of analysis they perform:

The *non-optimizing quotation compiler* compiles quotations to naive machine code very quickly. The non-optimizing quotation compiler is part of the VM.

The *optimizing word compiler* compiles whole words at a time while performing extensive data and control flow analysis. This provides greater performance for generated code, but incurs a much longer compile time. The optimizing compiler is written in Factor.

The optimizing compiler also trades off compile time for performance of generated code, so loading certain vocabularies might take a while. Saving the image after loading vocabularies can save you a lot of time that you would spend waiting for the same code to load in every coding session; see **Images** for information.

Most code you write will run with the optimizing compiler. Sometimes, the non-optimizing compiler is used, for example for listener interactions, or for running the quotation passed to **call(**.

Compiler errors

After loading a vocabulary, you might see a message like:

```
:errors - print 2 compiler errors
```

This indicates that some words did not pass the stack checker. Stack checker error conditions are documented in **Stack checker errors**, and the stack checker itself in **Stack effect checking**.

Words to view errors:

:errors (--)

:linkage (--)

Compiler errors are reported using the **Batch error reporting** mechanism, and as a result, they are also are shown in the **UI error list tool**.

Compiler specialization hints

Specialization hints help the compiler generate efficient code.

Specialization hints can help words which call a lot of generic words on the same object - perhaps in a loop - and in most cases, it is anticipated that this object is of a certain class, or even **eq?** to some literal. Using specialization hints, the compiler can be instructed to

compile a branch at the beginning of the word; if the branch is taken, the input object has the assumed class or value, and inlining of generic methods can take place.

Specialization hints are not declarations; if the inputs do not match what is specified, the word will still run, possibly slower if the compiled code cannot inline methods because of insufficient static type information.

In some cases, specialization will not help at all, and can make generated code slower from the increase in code size. The compiler is capable of inferring enough static type information to generate efficient code in many cases without explicit help from the programmer. Specializers should be used as a last resort, after profiling shows that a critical loop makes a lot of repeated calls to generic words which dispatch on the same class.

Type hints are declared with a parsing word:

HINTS:

The specialized version of a word which will be compiled by the compiler can be inspected:

specialized-def (word -- quot)

Calling the optimizing compiler

Normally, new word definitions are recompiled automatically. This can be changed:

disable-optimizer (--)

enable-optimizer (--)

More words can be found in **Compilation units**.

Compiler implementation

The **compiler** vocabulary, in addition to providing the user-visible words of the compiler, implements the main compilation loop.

Once compiled, a word is added to the assoc stored in the **compiled** variable. When compilation is complete, this assoc is passed to **modify-code-heap**.

The **compile-word** word performs the actual task of compiling an individual word. The process proceeds as follows:

The **frontend** word calls **build-tree**. If this fails, the error is passed to **deoptimize**. The logic for ignoring certain compile errors generated for inline words and macros is located here. If the error is not ignorable, it is added to the global **compiler-errors** assoc (see **Compiler errors**).

If the word contains a breakpoint, compilation ends here. Otherwise, all remaining steps execute until machine code is generated. Any further errors thrown by the compiler are not reported as compile errors, but instead are ordinary exceptions. This is because they indicate bugs in the compiler, not errors in user code.

The **frontend** word then calls **optimize-tree**. This produces the final optimized tree IR, and this stage of the compiler is complete.

The **backend** word calls **build-cfg** followed by **optimize-cfg** and a few other stages. Finally, it calls **generate**.

If compilation fails, the word is stored in the **compiled** assoc with a value of **f**. This causes the VM to compile the word with the non-optimizing compiler.

Calling **modify-code-heap** is handled not by the **compiler** vocabulary, but rather **compiler.units**. The optimizing compiler merely provides an implementation of the **recompile** generic word.

Batch error reporting

Some tools, such as the **Optimizing compiler**, **Unit testing** and **Help lint tool** need to report multiple errors at a time. Each error is associated with a source file, line number, and optionally, a definition. **Exception handling** cannot be used for this purpose, so the **source-files.errors** vocabulary provides an alternative mechanism. Note that the words in this vocabulary are used for implementation only; to actually list errors, consult the documentation for the relevant tools.

Source file errors inherit from a class:

source-file-error

Printing an error summary:

error-summary (--)

Printing a list of errors:

errors. (errors --)

Batch errors are reported in the **UI error list tool**.

Images

Factor has the ability to save the entire state of the system into an *image file*. The image contains a complete dump of all data and code in the current Factor instance.

save (--)

save-image (path --)

save-image-and-exit (path --)

To start Factor with a custom image, use the `-i=image` command line switch; see **Command line switches for the VM**.

One reason to save a custom image is if you find yourself loading the same libraries in every Factor session; some libraries take a little while to compile, so saving an image with those libraries loaded can save you a lot of time.

For example, to save an image with the web framework loaded,

```
USE: furnace
save
```

New images can be created from scratch:

Bootstrapping new images

The **Application deployment** tool creates stripped-down images containing just enough code to run a single application.

See also

Object memory tools

Bootstrapping new images

A new image can be built from source; this is known as *bootstrap*. Bootstrap is a two-step process. The first stage is the creation of a bootstrap image from a running Factor instance:

make-image (arch --)

The second bootstrapping stage is initiated by running the resulting bootstrap image:

```
./factor -i=boot.x86.32.image
```

This stage loads additional code, compiles all words, and dumps a final `factor.image`.

The bootstrap process can be customized with command-line switches.

See also

Command line switches for the VM, **Command line switches for bootstrap**

Command line arguments

Factor command line usage:
```
factor [VM args...] [script] [args...]
```

Zero or more VM arguments can be passed in, followed by an optional script file name. If the script file is specified, it will be run on startup using **run-script**. Any arguments after the script file are stored in the following variable, with no further processing by Factor itself:

command-line

Instead of running a script, it is also possible to run a vocabulary; this invokes the vocabulary's **MAIN:** word:
```
factor [system switches...] -run=<vocab name>
```

If no script file or `-run=` switch is specified, Factor will start **The listener** or **UI developer tools**, depending on the operating system.

As stated above, arguments in the first part of the command line, before the optional script name, are interpreted by to the Factor system. These arguments all start with a dash (`-`).

Switches can take one of the following three forms:

- `-foo` - sets the global variable `"foo "` to **t**
- `-no-foo` - sets the global variable `"foo "` to **f**
- `-foo=bar` - sets the global variable `"foo "` to `"bar "`

Command line switches for the VM
Command line switches for bootstrap
Command line switches for general usage

The raw list of command line arguments can also be obtained and inspected directly:

(command-line) (-- args)

There is a way to override the default vocabulary to run on startup, if no script name or `-run` switch is specified:

main-vocab-hook

Command line switches for the VM

A handful of command line switches are processed by the VM and not the library. They control low-level features.

`-i=image`	Specifies the image file to use; see **Images**
`-datastack=n`	Data stack size, kilobytes
`-retainstack=n`	Retain stack size, kilobytes
`-callstack=n`	Call stack size, kilobytes
`-young=`	Size of youngest generation (0), megabytes

`n`

`-aging=n` Size of aging generation (1), megabytes

`-tenured=n` Size of oldest generation (2), megabytes

`-codeheap=n` Code heap size, megabytes

`-callback=n` Callback heap size, megabytes

`-pic=n` Maximum inline cache size. Setting of 0 disables inline caching, > 1 enables polymorphic inline caching

`-securegc` If specified, unused portions of the data heap will be zeroed out after every garbage collection

If an `-i=` switch is not present, the default image file is used, which is usually a file named `factor.image` in the same directory as the Factor executable.

Command line switches for bootstrap

A number of command line switches can be passed to a bootstrap image to modify the behavior of the resulting image:

`-output-image=image` Save the result to `image`. The default is `factor.image`.

`-no-user-init` Inhibits the running of user initialization files on startup. See **Running code on startup**.

`-include=components...` A list of components to include (see below).

`-exclude=components...` A list of components to exclude.

`-ui-backend=backend` One of `x11`, `windows`, or `cocoa`. The default is platform-specific.

Bootstrap can load various optional components:

`math` Rational and complex number support.

`threads` Thread support.

`compiler` The compiler.

`tools` Terminal-based developer tools.

`help` The help system.

`help.handbook` The help handbook.

`ui` The graphical user interface.

`ui.tools` Graphical developer tools.

`io` Non-blocking I/O and networking.

By default, all optional components are loaded. To load all optional components except for a given list, use the `-exclude=` switch; to only load specified optional components, use the `-include=`.

For example, to build an image with the compiler but no other components, you could do:

```
./factor -i=boot.macosx-ppc.image -include=compiler
```

To build an image with everything except for the user interface and graphical tools,
`./factor -i=boot.macosx-ppc.image -exclude="ui ui.tools"`

To generate a bootstrap image in the first place, see **Bootstrapping new images**.

Command line switches for general usage

The following command line switches can be passed to a bootstrapped Factor image:

`-e=code` This specifies a code snippet to evaluate. If you want Factor to exit immediately after, also specify `-run=none`.

`-run=`*vocab* *vocab* is the name of a vocabulary with a **MAIN:** hook to run on startup, for example **listener**, **ui.tools** or **none**.

`-no-user-init` Inhibits the running of user initialization files on startup. See **Running code on startup**.

`-quiet` If set, **run-file** and **require** will not print load messages.

Running code on startup

Factor looks for three optional files in your home directory.

Bootstrap initialization file
Startup initialization file
Additional vocabulary roots file

The `-no-user-init` command line switch will inhibit loading running of these files.

If you are unsure where the files should be located, evaluate the following code:

```
USE: command-line
".factor-rc" rc-path print
".factor-boot-rc" rc-path print
```

Here is an example `.factor-boot-rc` which sets up GVIM editor integration:

```
USING: editors.gvim namespaces ;
"/opt/local/bin" \ gvim-path set-global
```

Bootstrap initialization file

The bootstrap initialization file is named `.factor-boot-rc`. This file can contain **require** calls for vocabularies you use frequently, and other such long-running tasks that you do not want to perform every time Factor starts.

A word to run this file from an existing Factor session:

run-bootstrap-init (--)

For example, if you changed `.factor-boot-rc` and do not want to bootstrap again, you can just invoke **run-bootstrap-init** in the listener.

Startup initialization file

The startup initialization file is named `.factor-rc`. If it exists, it is run every time Factor starts.

A word to run this file from an existing Factor session:

run-user-init (--)

Additional vocabulary roots file

The vocabulary roots file is named `.factor-roots` on Windows and `.factor-roots` on Unix. If it exists, it is loaded every time Factor starts. It contains a newline-separated list of **Vocabulary roots**.

A word to run this file from an existing Factor session:

load-vocab-roots (--)

Initialization and startup

When Factor starts, the first thing it does is call a word:

boot (--)

Next, initialization hooks are called:

do-startup-hooks (--)

Initialization hooks can be defined:

add-startup-hook (quot name --)

Corresponding shutdown hooks may also be defined:

add-shutdown-hook (quot name --)

The boot quotation can be changed:

startup-quot (-- quot)

set-startup-quot (quot --)

When quitting Factor, shutdown hooks are called:

do-shutdown-hooks (--)

System interface

Getting the path to the Factor VM and image:

vm (-- path)

image (-- path)

Getting a monotonically increasing nanosecond count:

nano-count (-- ns)

Exiting the Factor VM:

exit (n -- *)

Processor detection

Processor detection:

cpu (-- class)

Supported processors:

x86.32

x86.64

ppc

arm

Processor families:

x86

Operating system detection

Operating system detection:

os (-- class)

Supported operating systems:

freebsd

linux

macosx

openbsd

netbsd

solaris

wince

winnt

Operating system families:

bsd

unix

windows

VM memory layouts

The words documented in this section do not ever need to be called by user code. They are documented for the benefit of those wishing to explore the internals of Factor's implementation.

Type numbers
Tagged pointers
Sizes and limits
Bootstrap support

Type numbers

Corresponding to every built-in class is a built-in type number. An object can be asked for its built-in type number:

tag (object -- n)

Built-in type numbers can be converted to classes, and vice versa:

type>class (n -- class)

type-number (class -- n)

num-types

See also

Built-in classes

Tagged pointers

Every pointer stored on the stack or in the heap has a *tag*, which is a small integer identifying the type of the pointer. If the tag is not equal to one of the two special tags, the remaining bits contain the memory address of a heap-allocated object. The two special tags are the **fixnum** tag and the **f** tag.

Words for working with tagged pointers:

tag-bits

tag-mask

The Factor VM does not actually expose any words for working with tagged pointers directly. The above words operate on integers; they are used in the bootstrap image generator and the optimizing compiler.

Sizes and limits

Processor cell size:

cell (-- n)

cells (m -- n)

cell-bits (-- n)

Range of integers representable by **fixnum**s:

most-negative-fixnum (-- n)

most-positive-fixnum (-- n)

Maximum array size:
max-array-capacity (-- n)

Bootstrap support

Processor cell size for the target architecture:
bootstrap-cell (-- n)

bootstrap-cells (m -- n)

bootstrap-cell-bits (-- n)

Range of integers representable by **fixnum**s of the target architecture:
bootstrap-most-negative-fixnum (-- n)

bootstrap-most-positive-fixnum (-- n)

Maximum array size for the target architecture:
bootstrap-max-array-capacity (-- n)